jpf-nhandler: Automated Handling of Native Calls in JPF

Nastaran Shafiei

Fujitsu Laboratories of America, Inc.

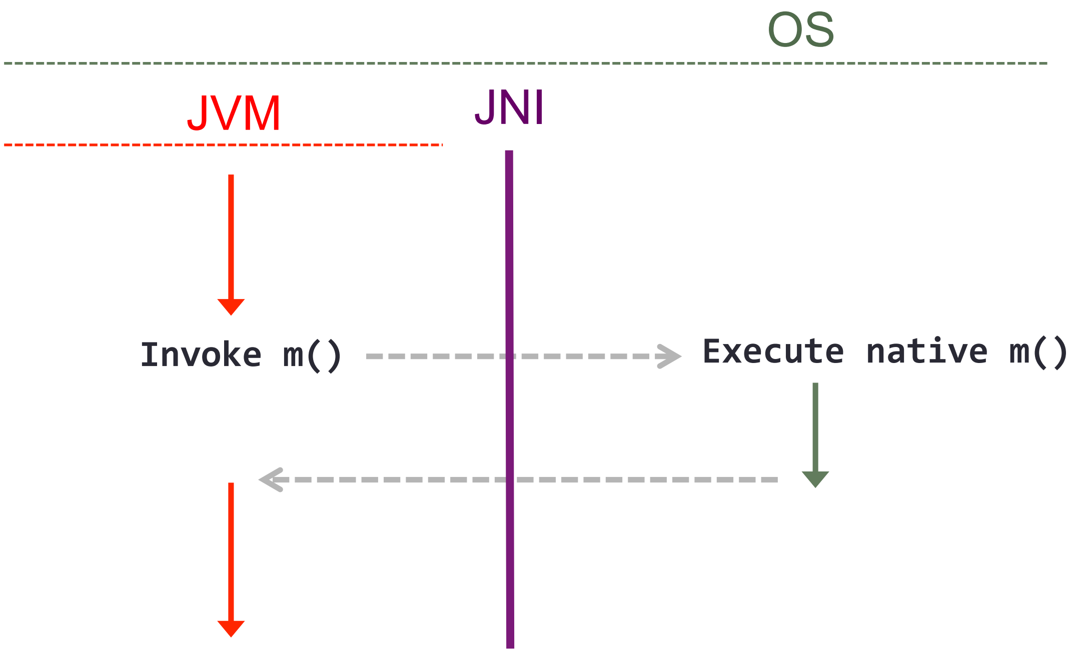
DisCoVeri group, York University, Toronto

Outline

- Background
- Motivation
- A technique to handle native calls
- More applications of the proposed technique
- Limitations
- Statistics
- Future work

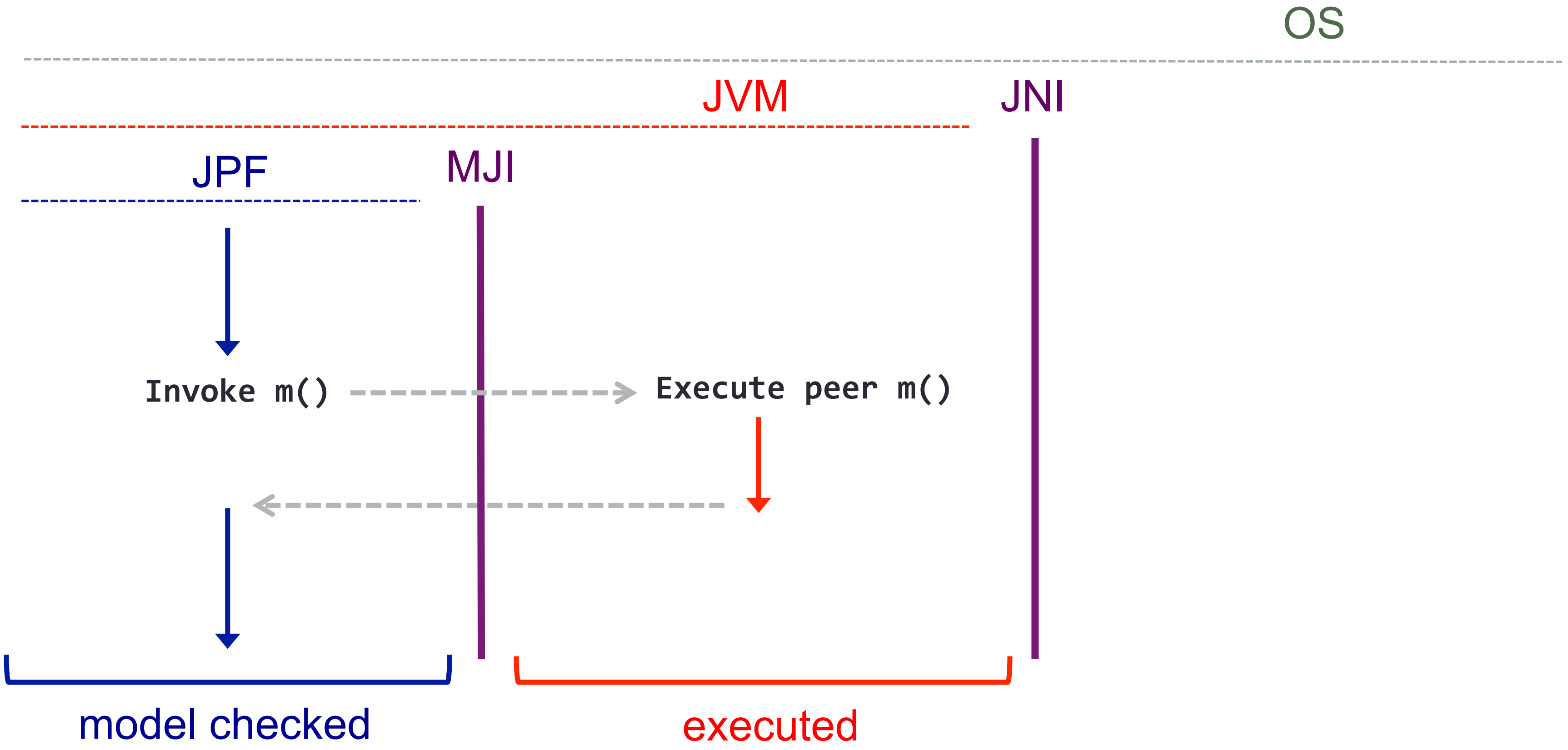
Native Calls

- Native calls: functions that are invoked from Java code and written in other languages (e.g. C, C++, assembly)
- JVM provides Java Native Interface (JNI) to delegate the execution from the Java level to the native level



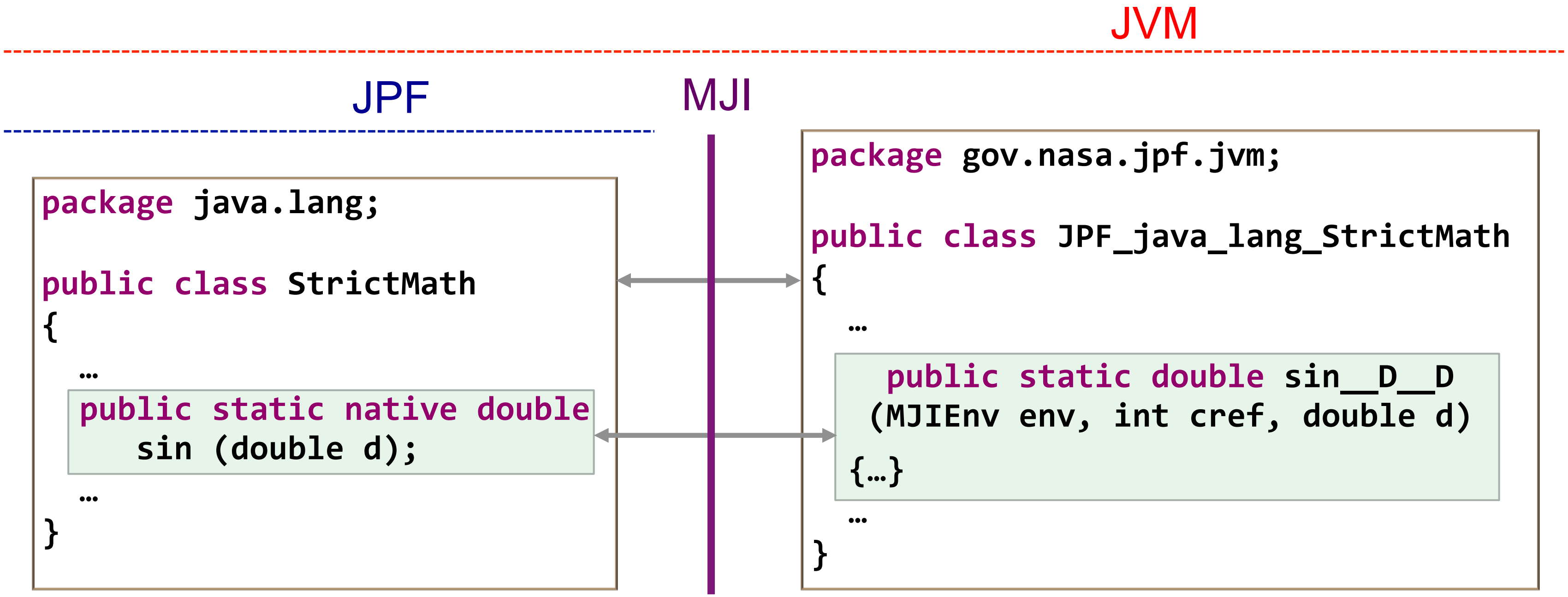
Native Calls in JPF

- JPF uses Model Java Interface (MJI) to handle native calls



Native Peers

- Native peers paly the key role in MJI
- They are executed by the host JVM
- A specific name pattern is used to map a class in JPF to a native peer

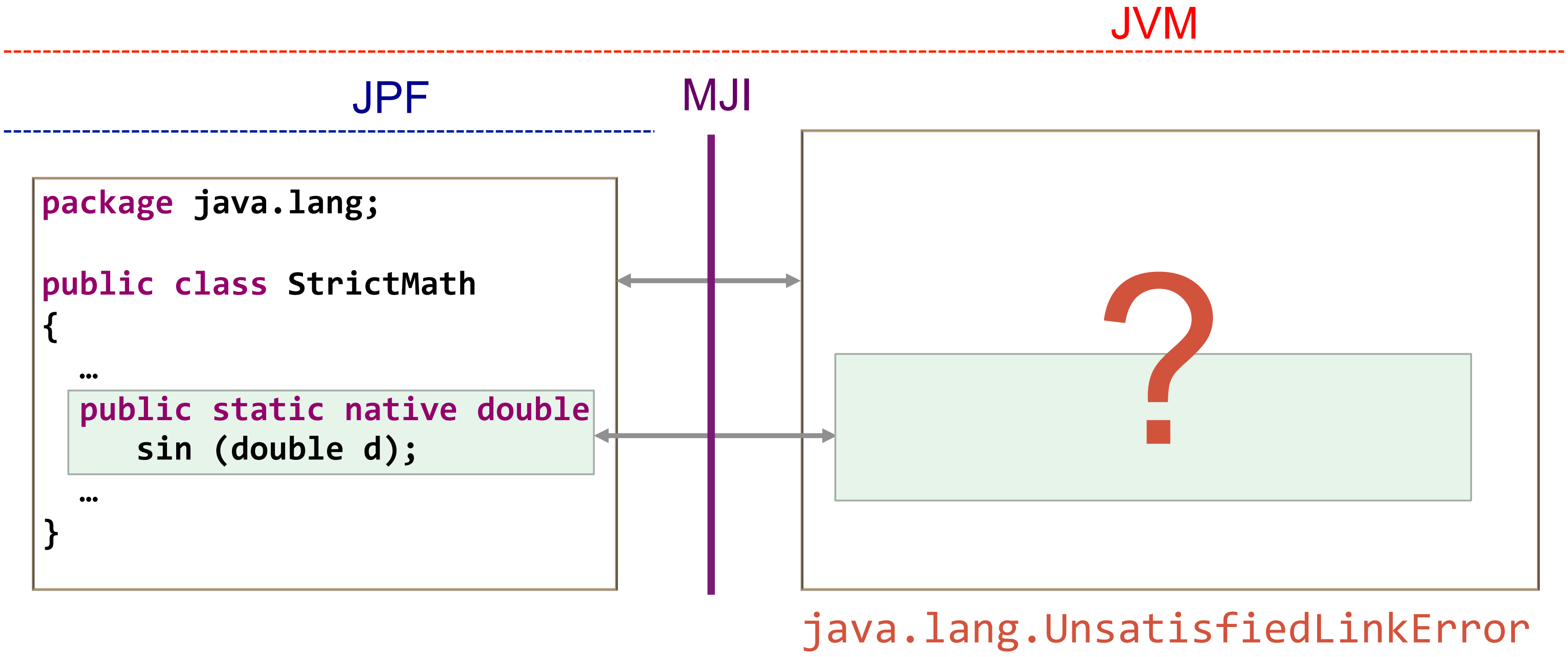


```
package java.lang;

public class StrictMath
{
  …
  public static native double
     sin (double d);
  …
}
```

```
package gov.nasa.jpf.jvm;

public class JPF_java_lang_StrictMath
{
  …
    public static double sin__D__D
   (MJIEnv env, int cref, double d)
  {…}
  …
}
```

Native Peers

- Native peers paly the key role in MJI
- They are executed by the host JVM
- A specific name pattern is used to map a class in JPF to a native peer



```
package java.lang;

public class StrictMath
{
   …
   public static native double
      sin (double d);
   …
}
```

java.lang.UnsatisfiedLinkError

Handling Native Calls Is Hard

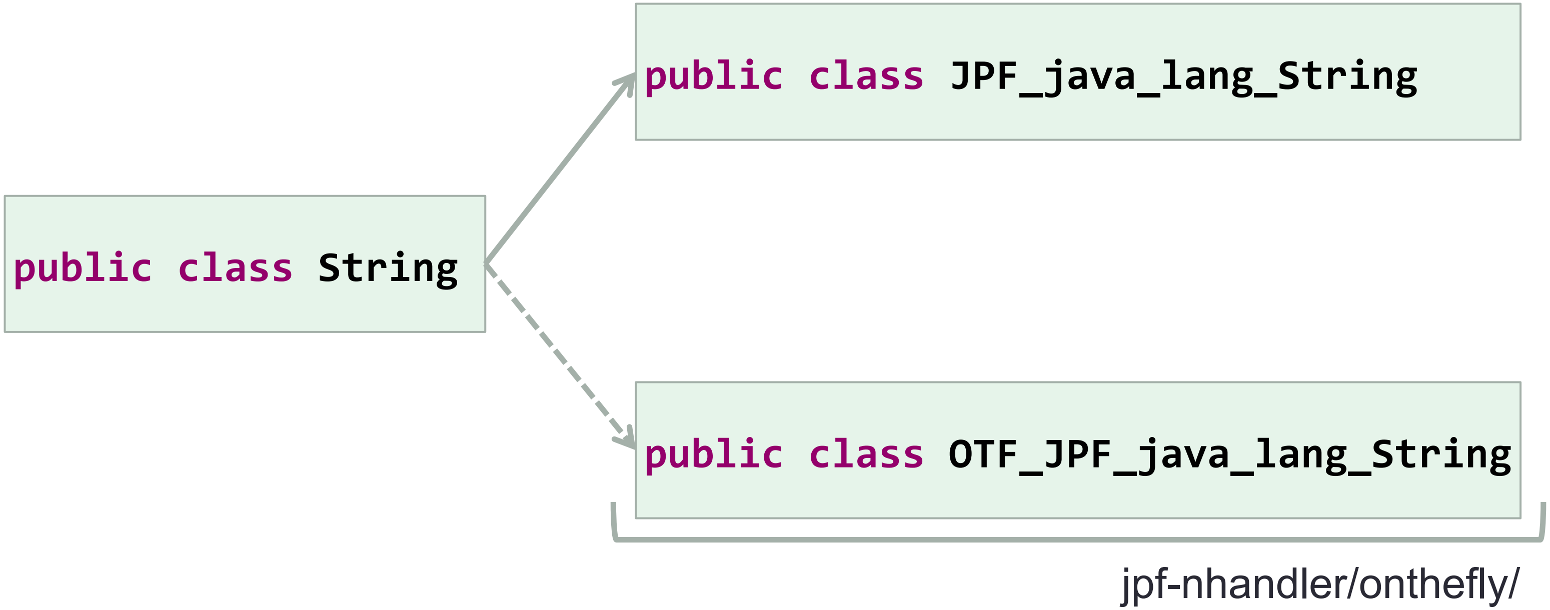
- Has to be done manually
- Requires knowledge from the internal structure of JPF

```
public static int NATIVE_PEER_METHOD
                 (MJIEnv env, int cref, int arg1,. . .)
{
   ElementInfo ei = env.getElementInfo(cref);
   ClassInfo JPFCl = ei.getClassInfo();
   . . .
 }
```

- The native method may not be recognizable
- Behavior of the native method may not be clear

jpf-nhandler

- An extension of JPF that handles native calls automatically
- Creates and maps native peers on-the-fly and on-demand



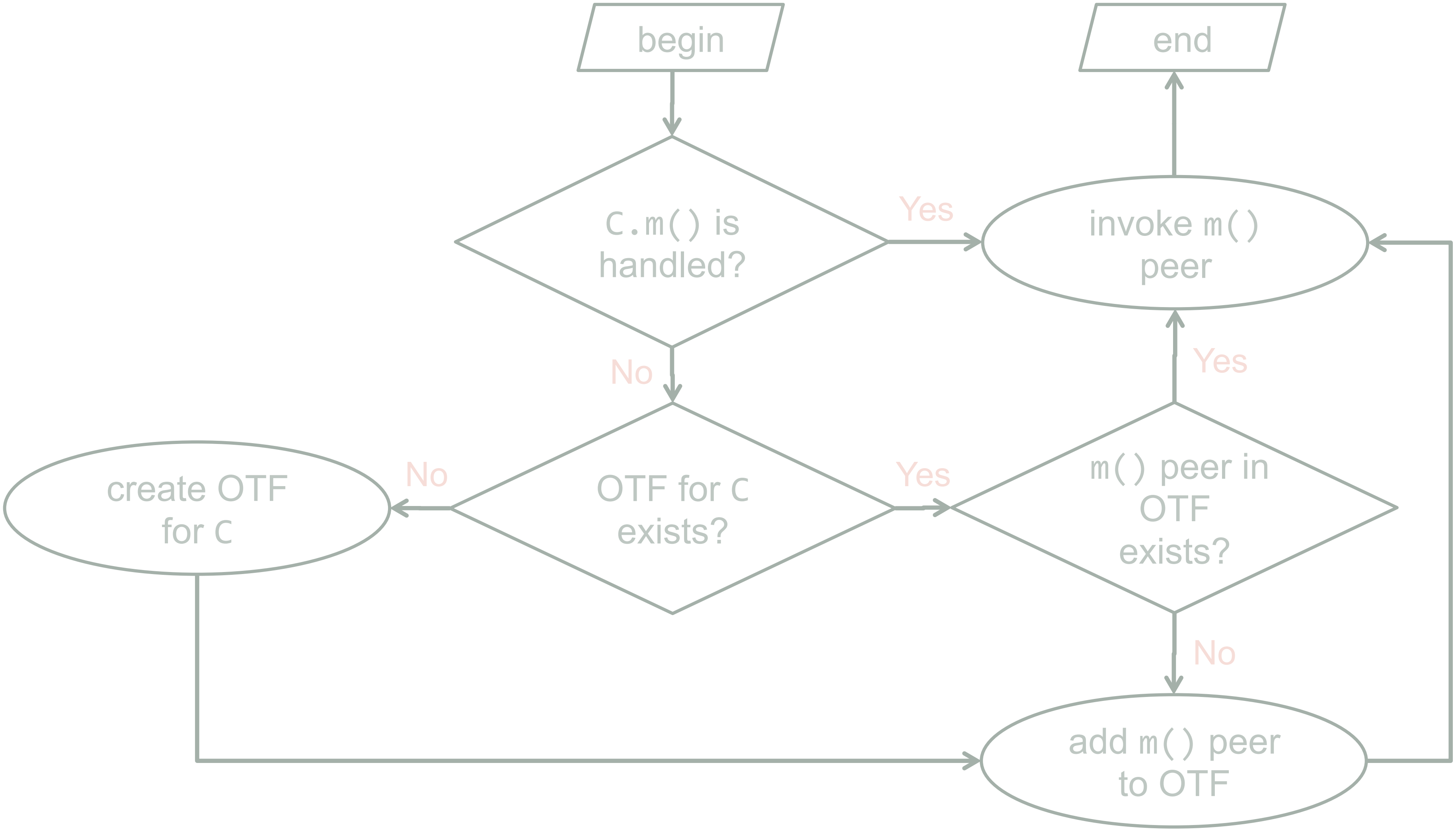
```
public class String
```

```
public class JPF_java_lang_String
```

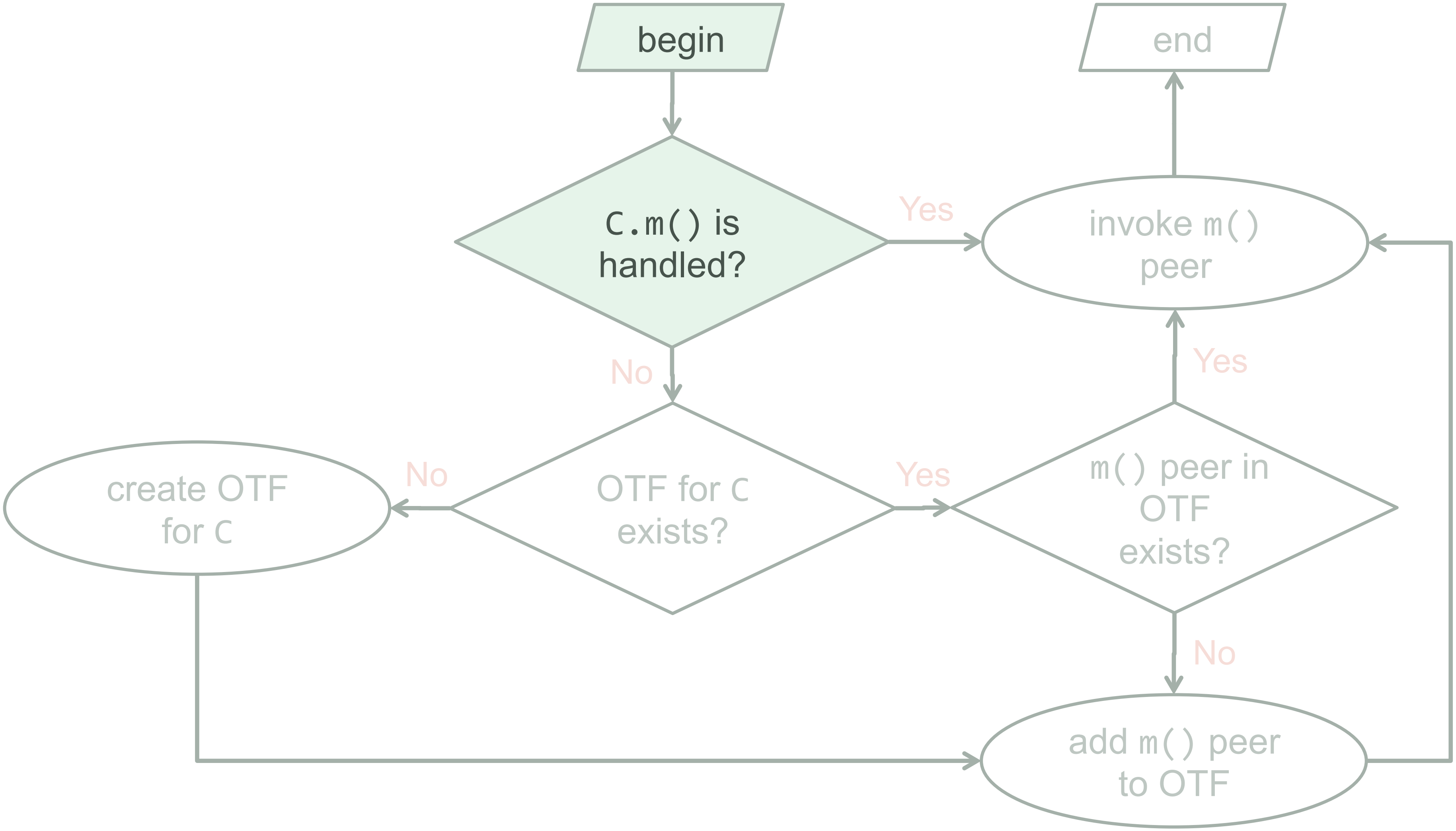
```
public class OTF_JPF_java_lang_String
```

jpf-nhandler/onthefly/

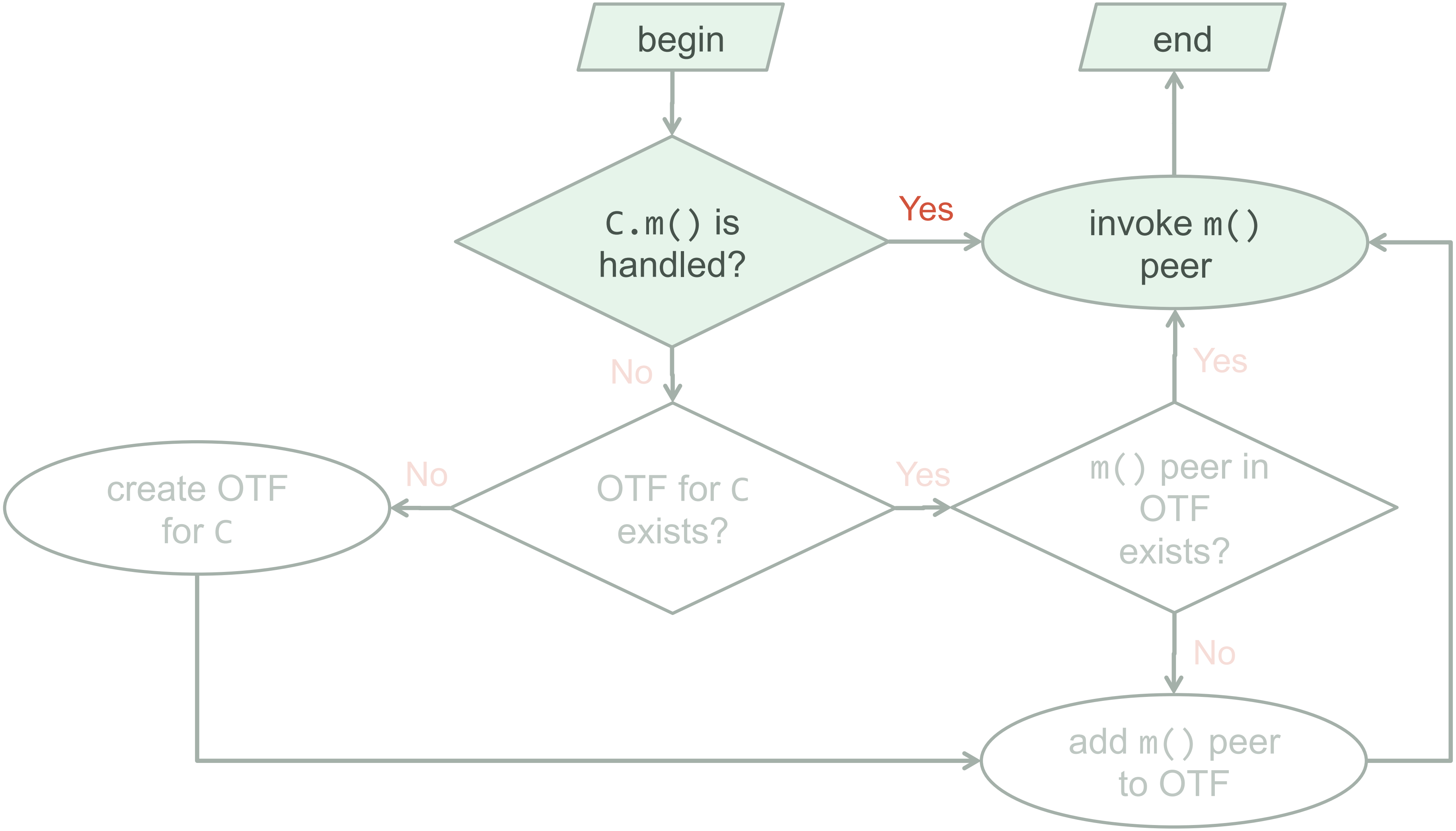
Creating Peers On-the-fly



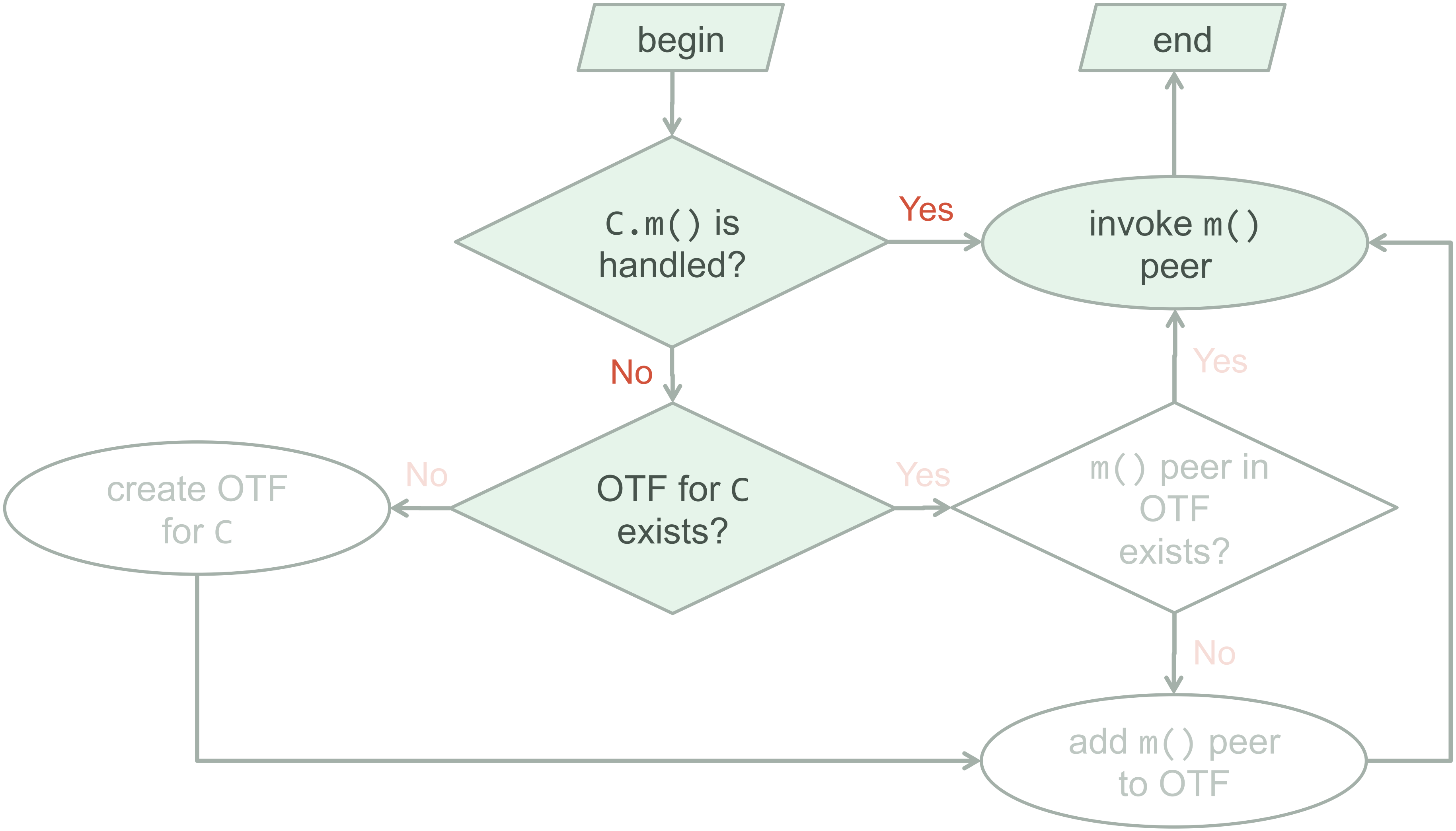
Creating Peers On-the-fly



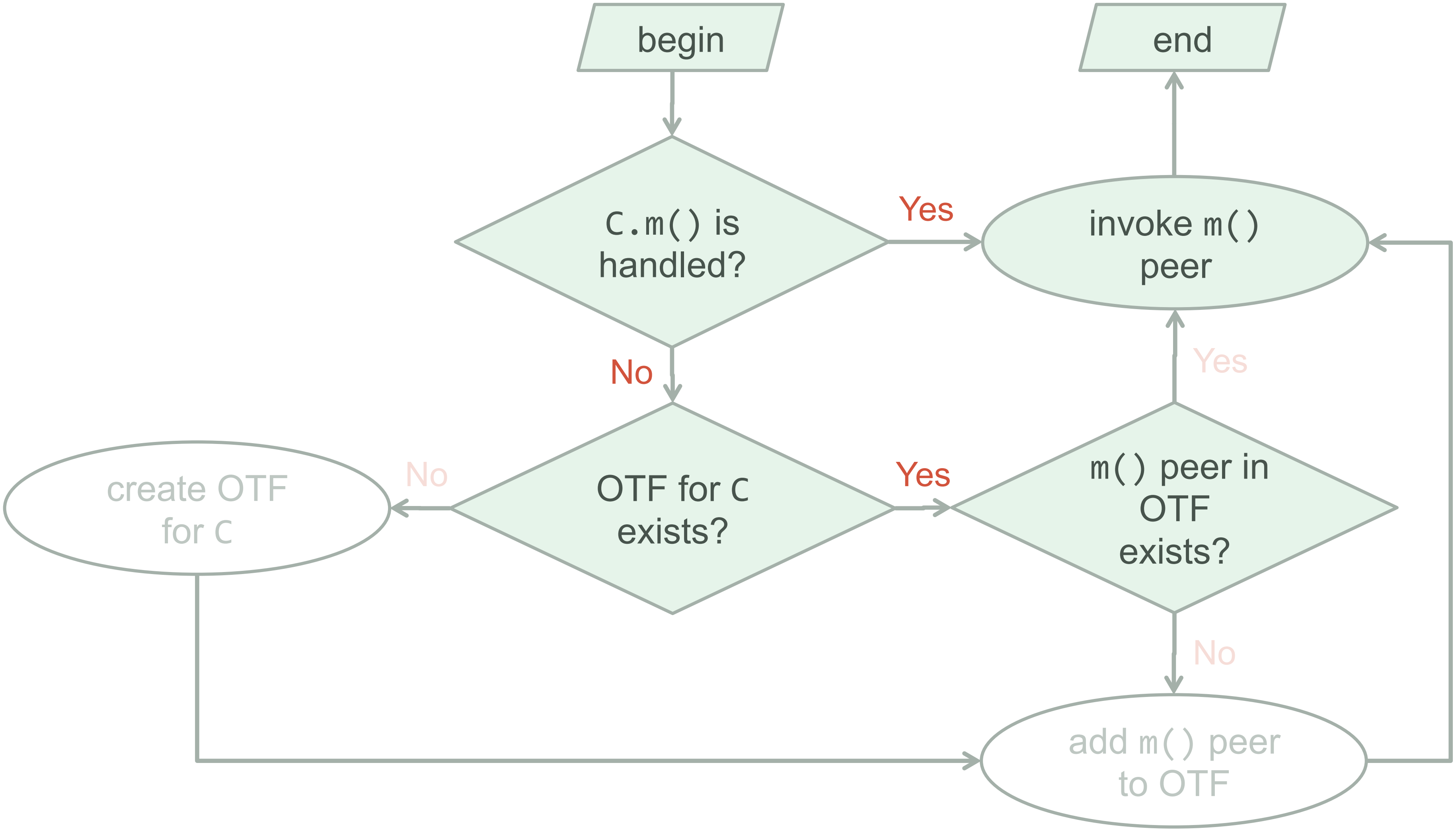
Creating Peers On-the-fly



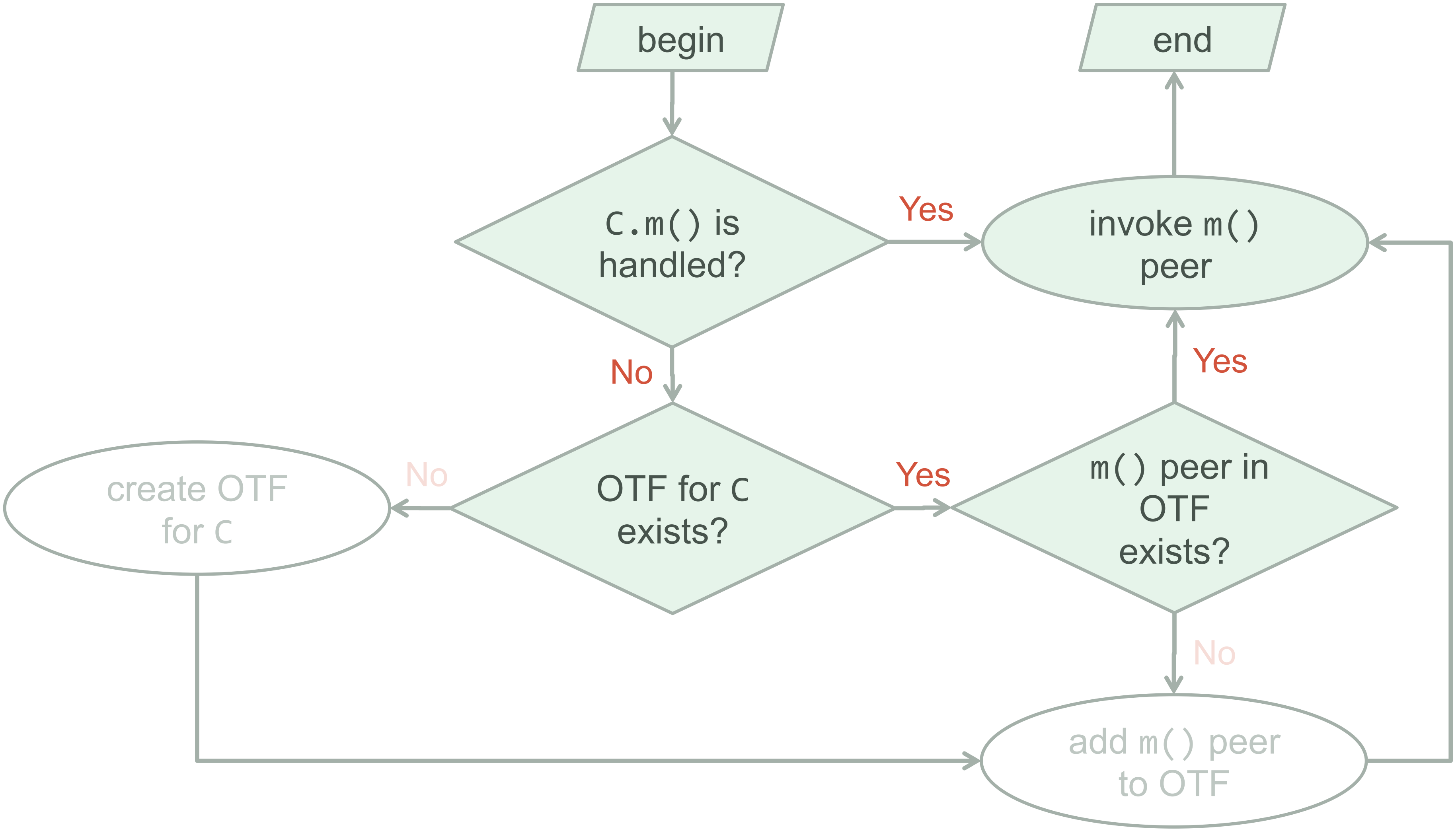
Creating Peers On-the-fly



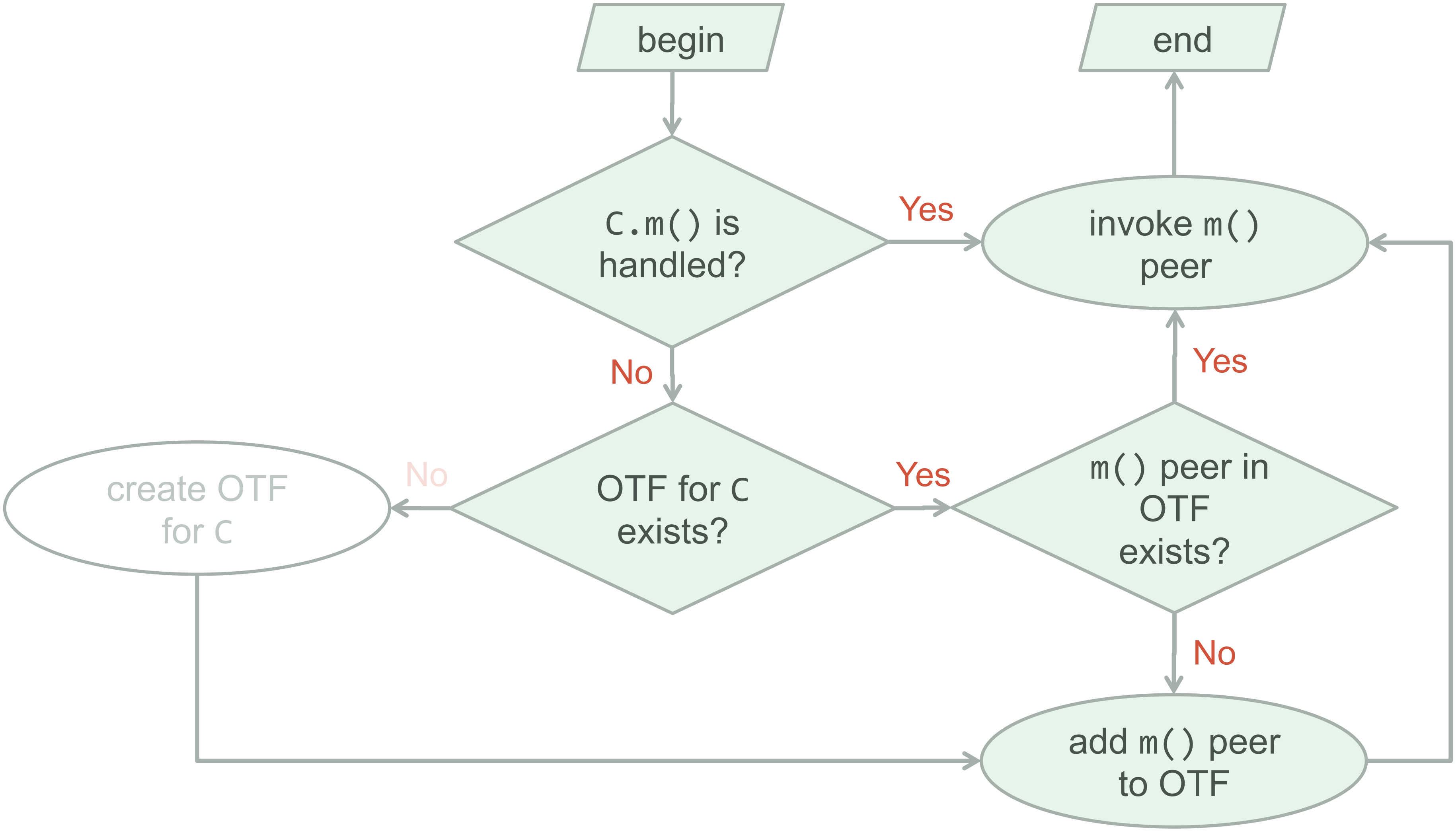
Creating Peers On-the-fly



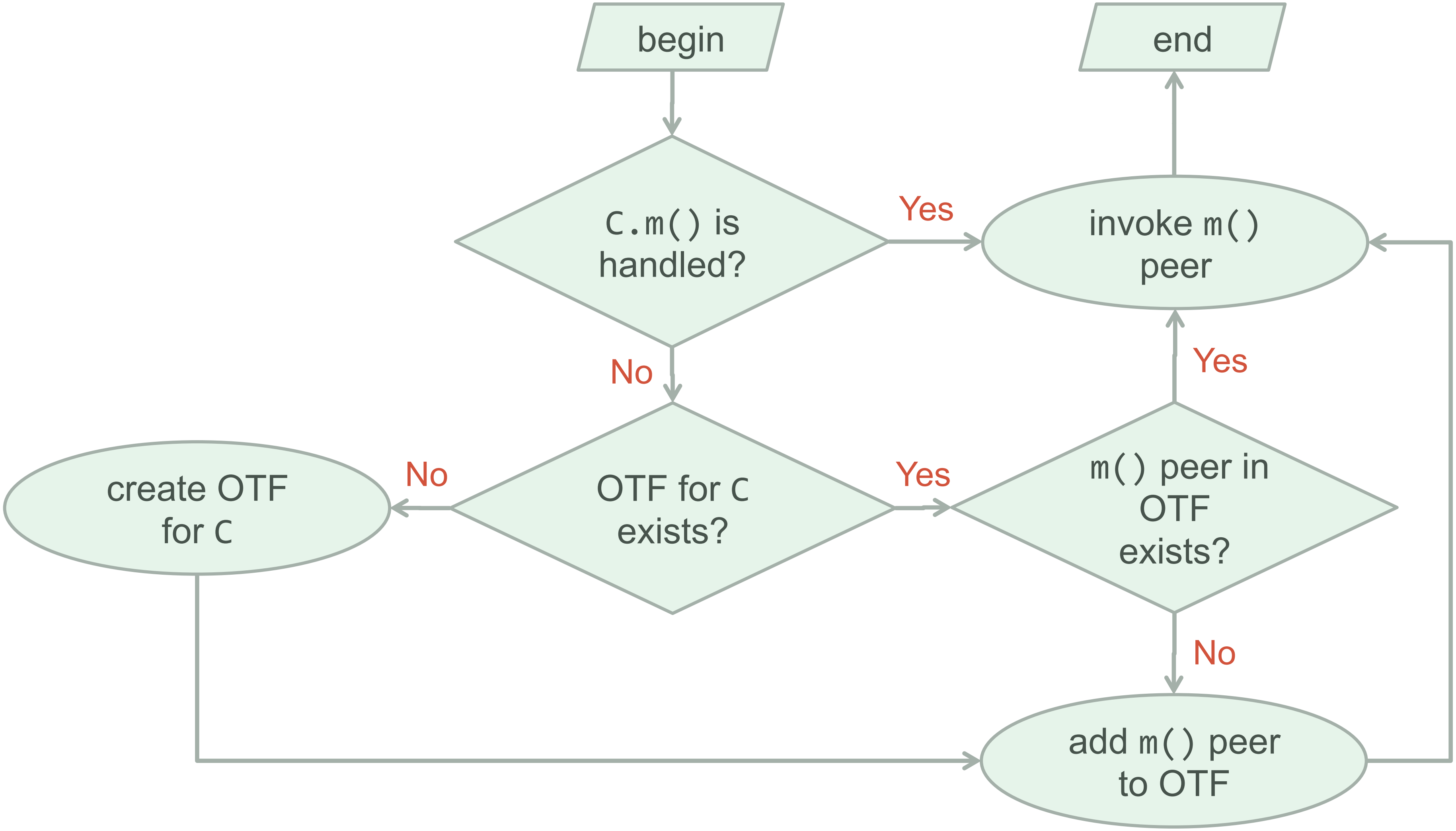
Creating Peers On-the-fly



Creating Peers On-the-fly

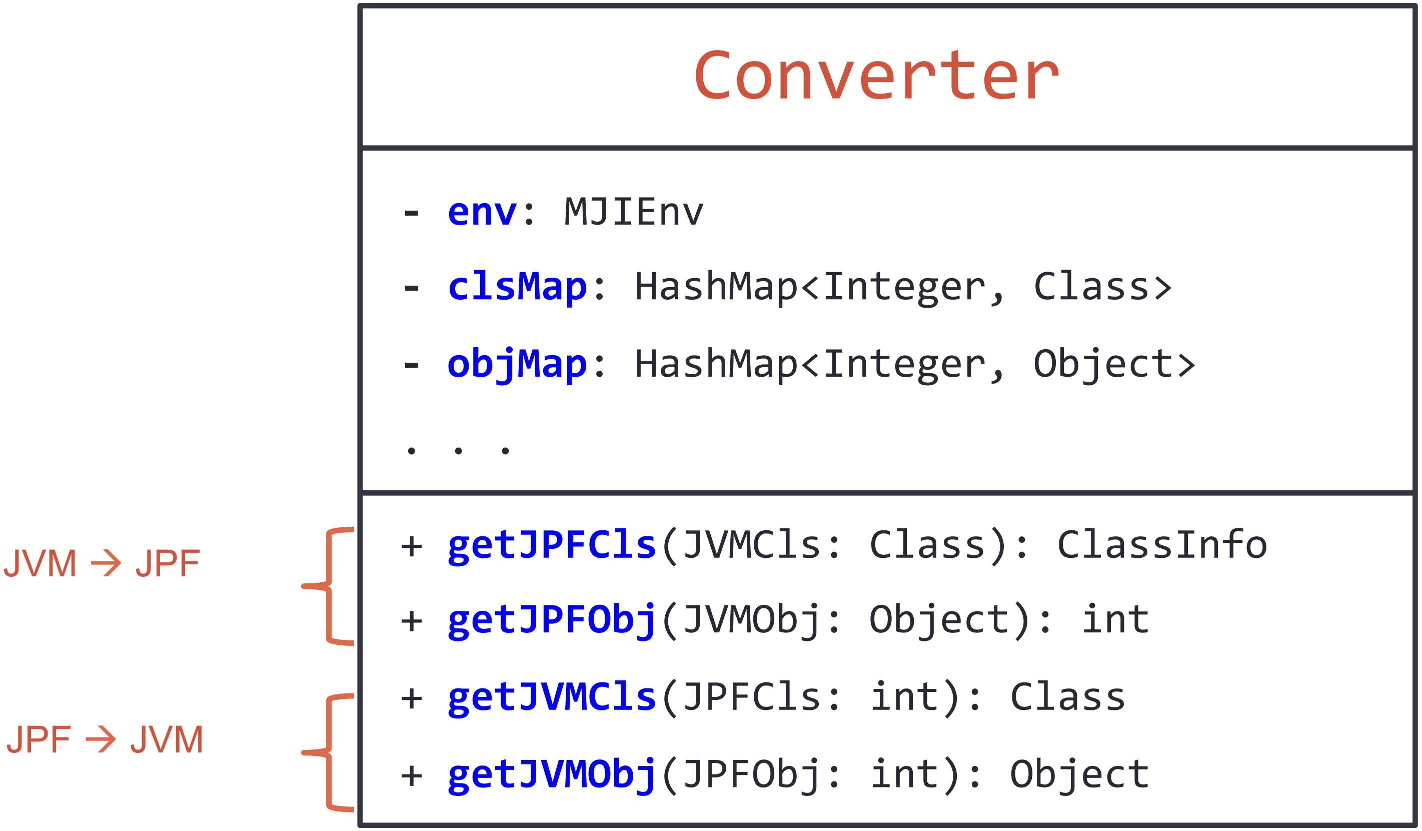


Creating Peers On-the-fly



Converter

- Converter is a main component of jpf-nhandler
- It converts objects and classes between JPF and the host JVM



Converter

```
- env: MJIEnv
- clsMap: HashMap<Integer, Class>
- objMap: HashMap<Integer, Object>
. . .
```

```
+ getJPFCls(JVMCls: Class): ClassInfo
+ getJPFObj(JVMObj: Object): int
+ getJVMCls(JPFCls: int): Class
+ getJVMObj(JPFObj: int): Object
```

getJVMCls(int JPFCls)

```
public Class getJVMCls(int JPFCls)
{
  JVMCls ← Class object representing JPFCls
  Add JVMCls to clsMap

  for each static field, f, of JVMCls
    if f is primitive
      JVMCls.f ← JPFCls.f
    else
      JVMCls.f ← getJVMObj(JPFCls.f)

  return JVMCls
}
```

getJVMObj(int JPFObj)

```
public Class getJVMObj(int JPFObj)
{
  JVMCls ← getJVMCls(class of JFPObj)
  JVMObj ← new instance of JVMCls
  Add JVMObj to objMap

  for each non-static field, f, of JVMCls
    if f is primitive
      JVMObj.f ← JPFObj.f
    else
      JVMObj.f ← getJVMObj(JPFObj.f)

  return JVMObj
}
```

Example

```
Class C {
  . . .
  native C2 m(C1 o1);
}
```

Consider jpf-nhandler is running on the following code snippet:

```
C o = new C();
o.m(o1);
```

Peer for o.m(o1)

```
public static int m__LC1_2__LC2_2 (MJIEnv env, int o, int o1)
{

    step 1: Capture objects & Classes in JVM

    step 2: Invoke the host JVM method m

    step 3: Convert the return value to a JPF object

    step 4: Apply changes to the JPF environment

}
```

Step 1: Captures Objects & Classes in JVM

- Root items to be converted from JPF to JVM:
  - Object/Class invoking the native method
  - Objects sent as an arguments to the native method and their classes

- `o.m(o1)`
  - `Object caller = converter.getJVMObj(o);`
  - `Object arg = converter.getJVMObj(o1);`

Peer for `o.m(o1)`

```
public static int m__LC1_2__LC2_2 (MJIEnv env, int o, int o1)
{

    step 1: Capture objects & Classes in JVM

    step 2: Invoke the host JVM method m

    step 3: Convert the return value to a JPF object

    step 4: Apply changes to the JPF environment

}
```

Step 2: Invoke the Native Method m

- Using reflection to get the Method object & invoke it

- `o.m(o1)`
  - `Method method = caller.getClass().getDeclaredMethod(”m”,…);`
  - `Object returnValue = method.invoke(caller,Object[]{arg});`

Peer for `o.m(o1)`

```
public static int m__LC1_2__LC2_2 (MJIEnv env, int o, int o1)
{

     step 1: Capture objects & Classes in JVM

     step 2: Invoke the host JVM method m

     step 3: Convert the return value to a JPF object

     step 4: Apply changes to the JPF environment

}
```

Step 3: Convert the Return Value to JPF Object

- The primitive types format in JPF is the same as their format in the host JVM

- If the return value is of non-primitive type, it is converted to a JPF object

- `o.m(o1)`
  - `int ret = converter.getJPFObj(returnValue);`

Peer for `o.m(o1)`

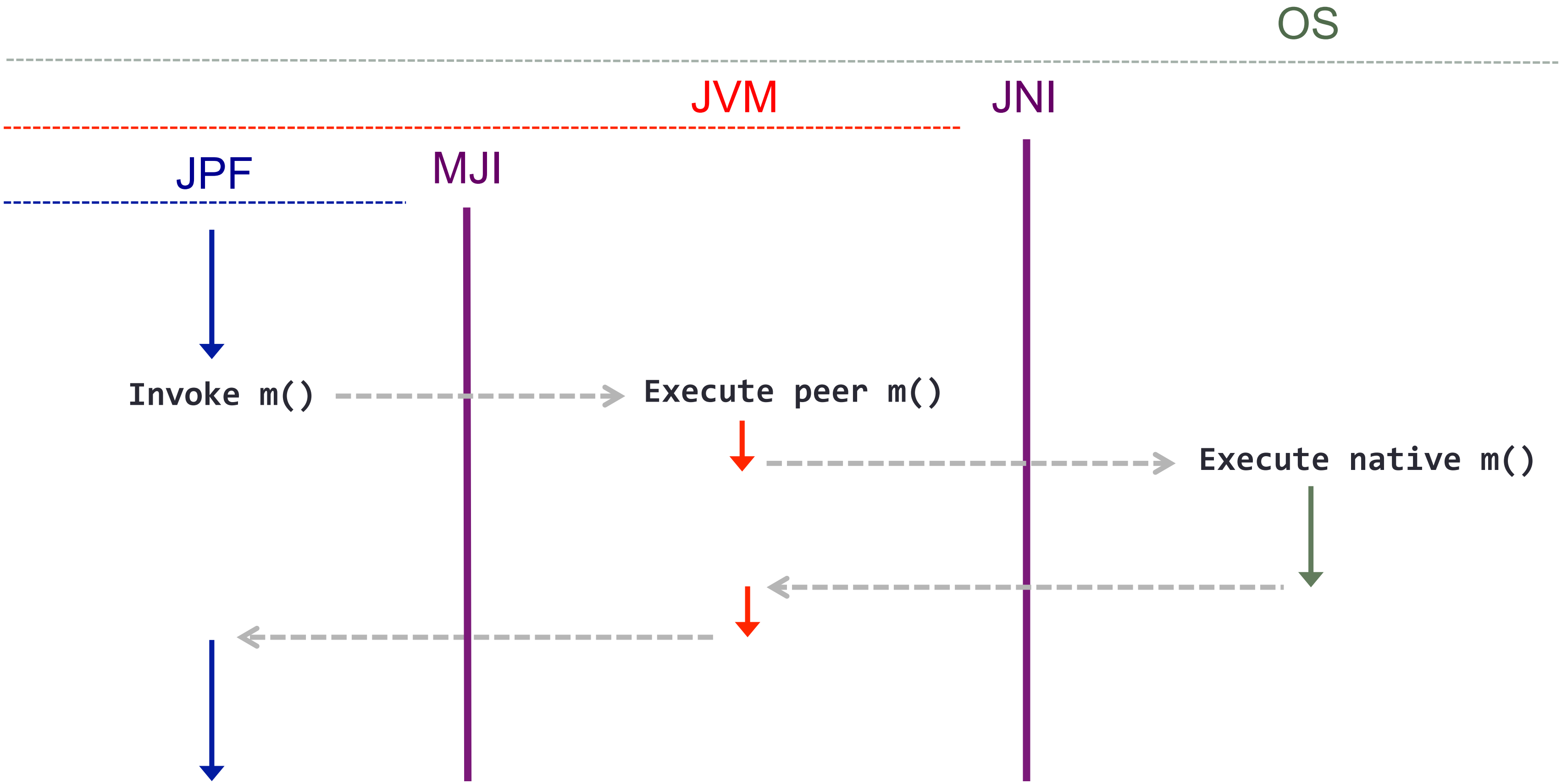
```
public static int m__LC1_2__LC2_2 (MJIEnv env, int o, int o1)
{

     step 1: Capture objects & Classes in JVM

     step 2: Invoke the host JVM method m

     step 3: Convert the return value to a JPF object

     step 4: Apply changes to the JPF environment

}
```

Step 4: Apply Changes to the JPF Environment

- Visibility of attributes from native methods is similar to the visibility of attributes from non-native methods
- The native method can access
  - Any static attributes
  - Non-static attributes declared in
    - Object invoking the native method
    - Objects sent as arguments to the native method
- `o.m(o1)`
  - Updating the JPF objects o & o1 and their classes

Execution Pattern

- To handle a native method, jpf-nhandler delegates the execution from the JPF level to the native level



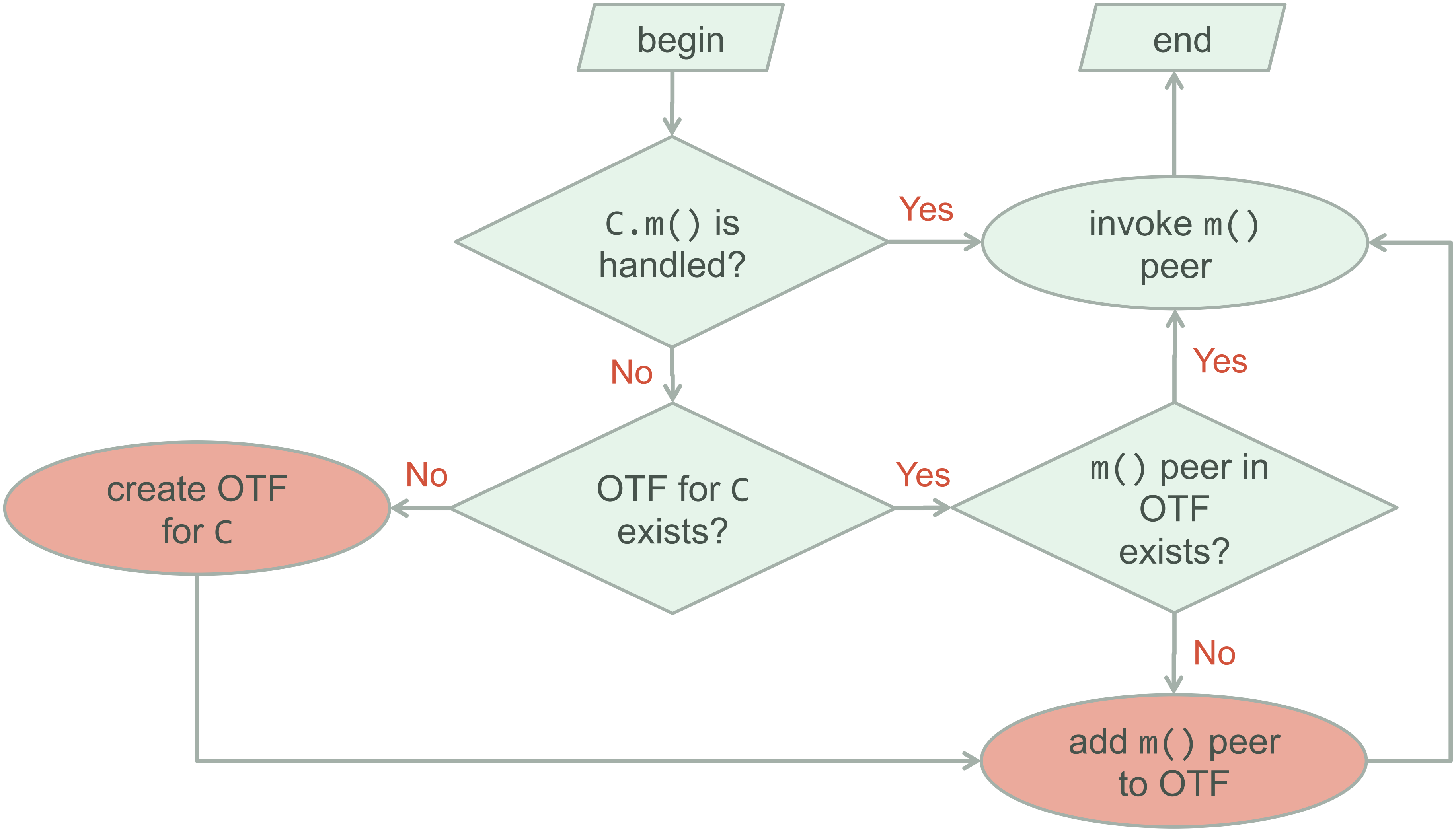
Ways to handle native method

- User can choose between the following two options:

1. Execute the steps within OTF peers
   - Requires creating, extending, and loading the classes
   - Source code can be obtained and edited by decompiling OTF peer (e.g. skipping the 4th step for certain methods)
   - Is the suggested option for frequently used native calls

2. Execute the steps outside of OTF peers
   - Skips creating, extending, and loading the classes

Handle Within OTF peer



Applications of jpf-nhandler

- It is not specific to native methods
    - lower down the execution from JPF to the host JVM for non-native methods

- Applications
    - Delegating native methods
        - To avoid `java.lang.UnsatisfiedLinkError`
    - Delegating non-native methods
        - Increases the performance by reducing the size of the state space
        - Simplifies the traces

Configuration

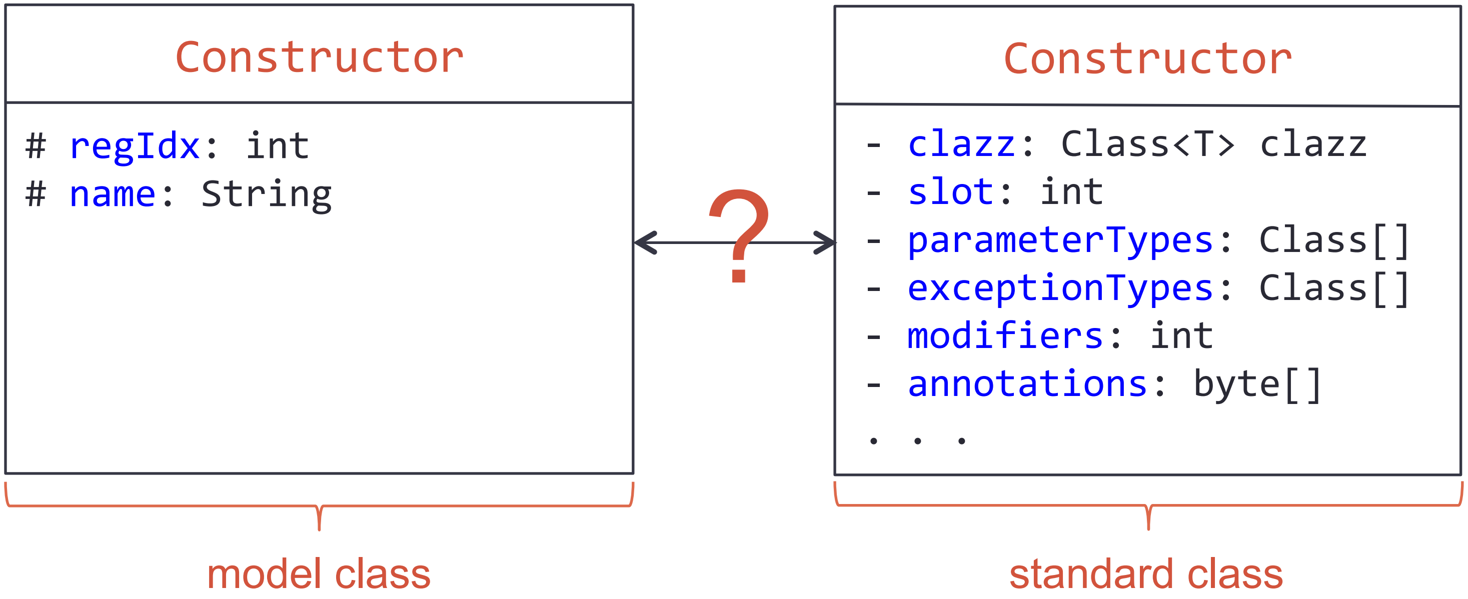
- Specifying the way to handle the methods
  - Within OTF peers
  - Skip creating OTF peers
- Specifying the methods to be handled
  - All unhandled native methods (default)
  - Certain native methods
  - Certain non-native methods
- Avoiding jpf-nhandler to delegate the execution for certain classes and methods
- Skipping the execution of certain methods

Limitations

- Correctness issues
  - Applying jpf-nhandler on certain system classes affects the system consistency (e.g. `Class, Thread, ThreadGroup`)
  - State of an object should be identified by the same fields and superclasses in host VM and JPF classes - guaranteed if the corresponding class hierarchies don't include modeled classes
  - The method execution should rely on caller and arguments & its side effects should be observable from return value, arguments, and caller
- Performance issues
  - Overhead due to irrelevant object conversion
  - Multiple instances of a JVM object retrieved from the object pool can be created in JPF

Limitations

- There should be a one-to-one correspondence between model class fields and the fields of the class in the Java standard library



Constructor

```
# regIdx: int
# name: String
```

?

Constructor

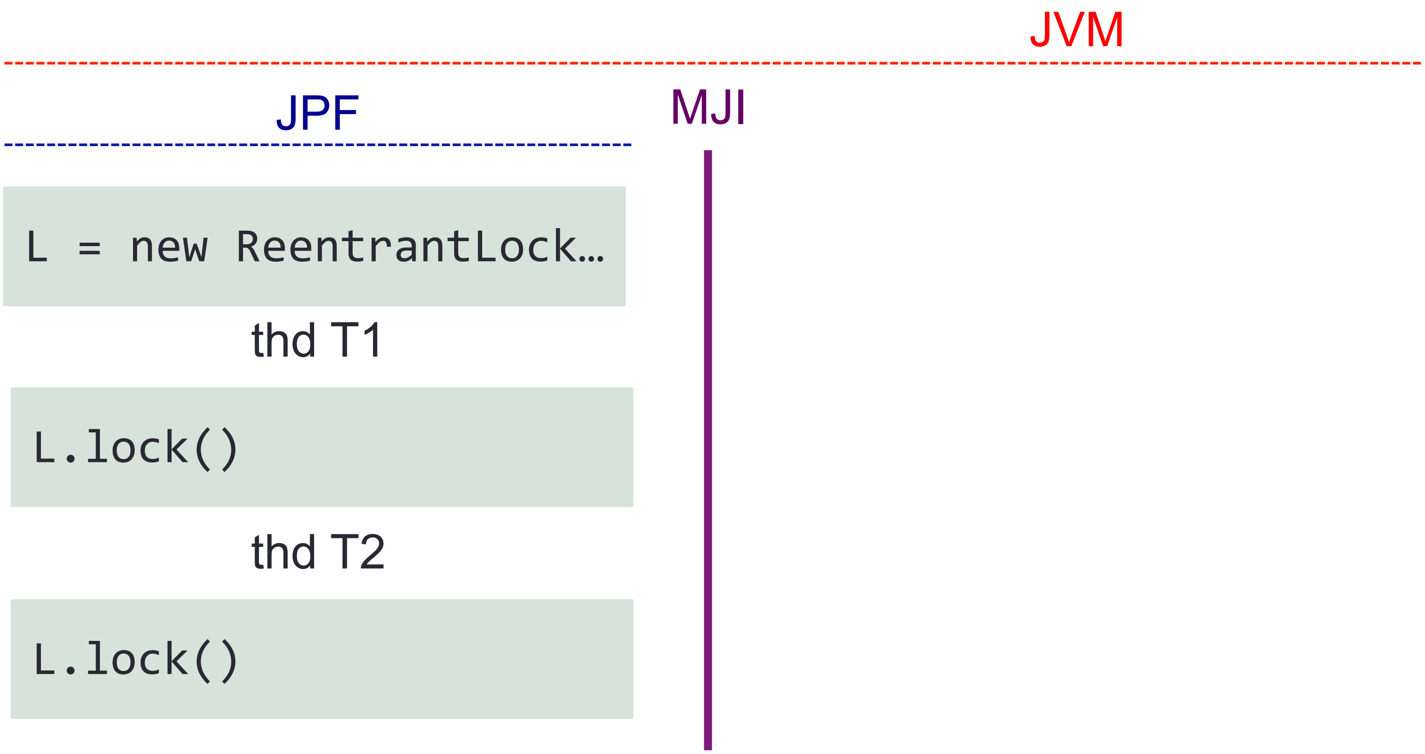
```
- clazz: Class<T> clazz
- slot: int
- parameterTypes: Class[]
- exceptionTypes: Class[]
- modifiers: int
- annotations: byte[]
. . .
```

model class

standard class

Limitations

- The execution of the method should only depend on the state of the caller object/class and the method's arguments
  - `ReentrantLock.lock()` depends on the current thread



JVM

JPF

MJI

```
L = new ReentrantLock…
```

thd T1

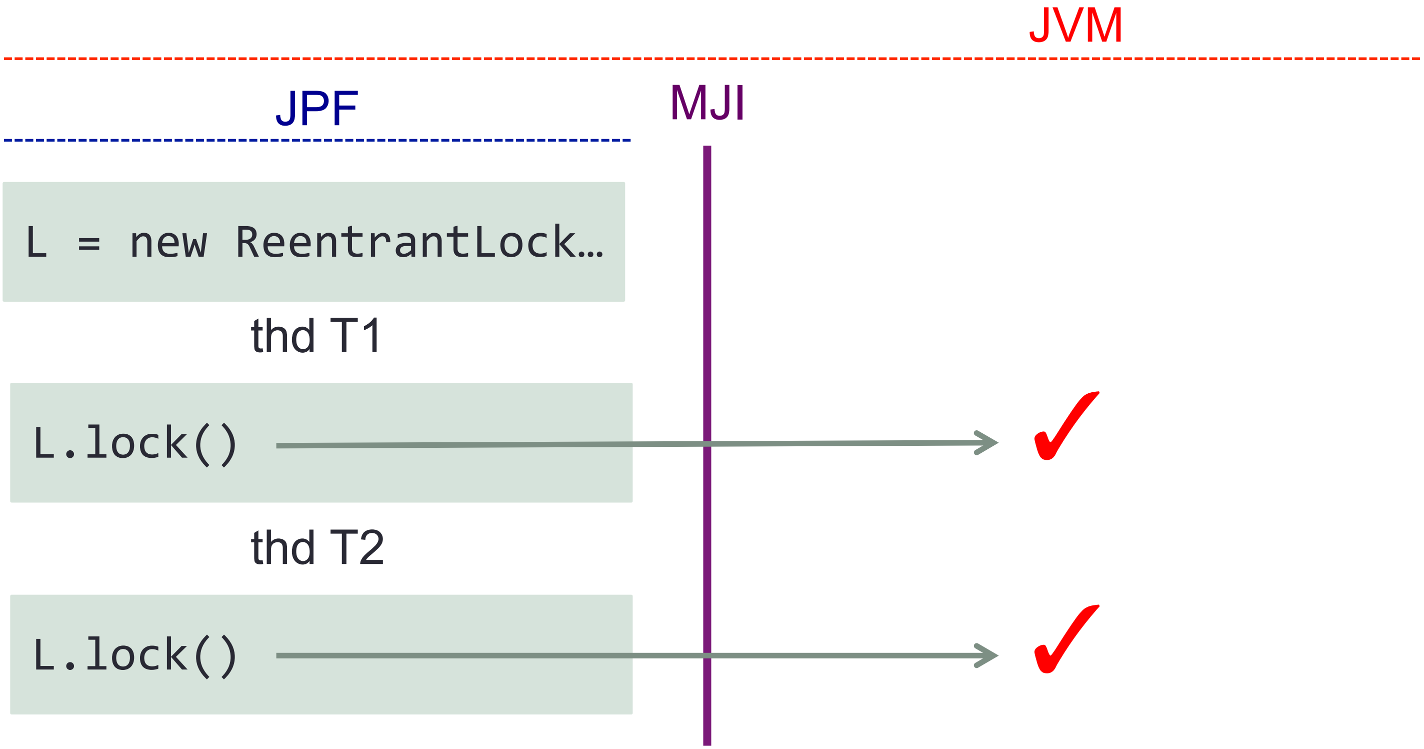
```
L.lock()
```

thd T2

```
L.lock()
```

Limitations

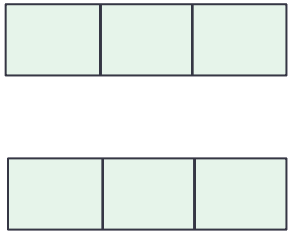
- The execution of the method should only depend on the state of the caller object/class and the method's arguments
  - `ReentrantLock.lock()` depends on the current thread



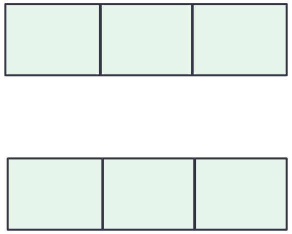
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



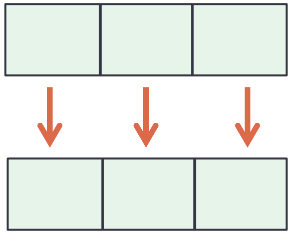
- Handled in jpf-nhandler



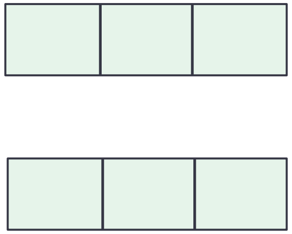
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



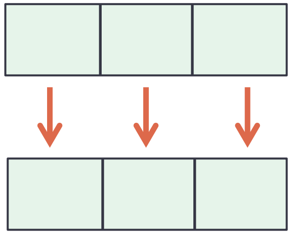
- Handled in jpf-nhandler



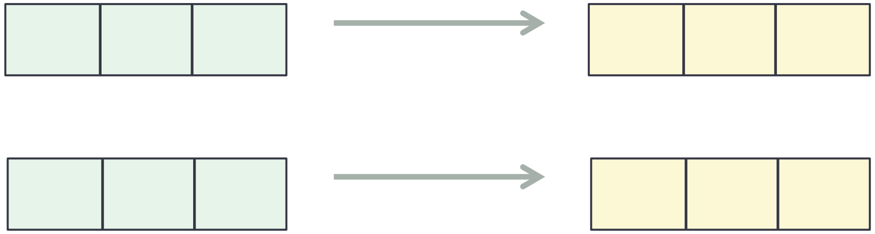
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



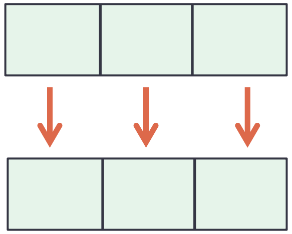
- Handled in jpf-nhandler



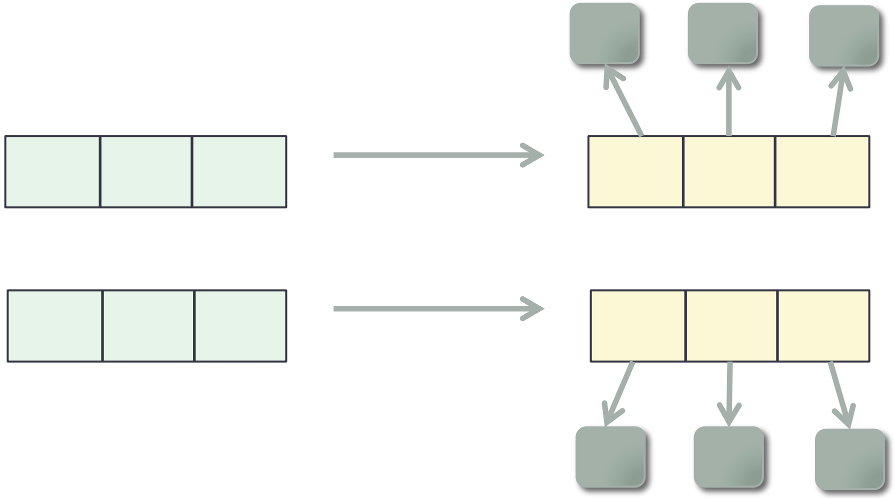
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



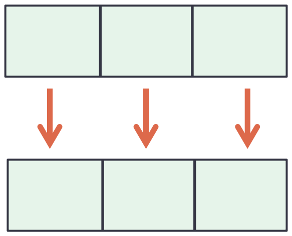
- Handled in jpf-nhandler



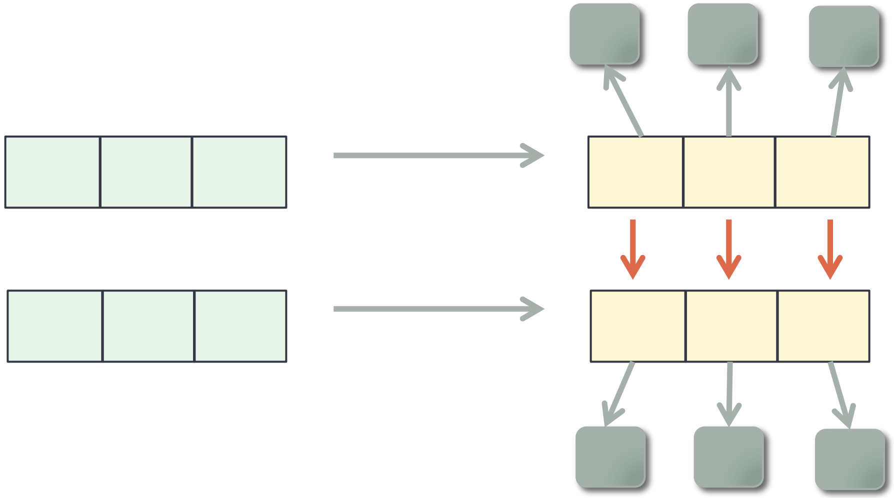
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



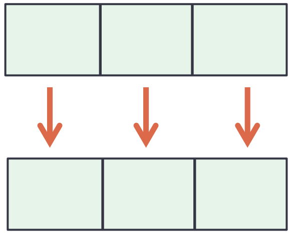
- Handled in jpf-nhandler



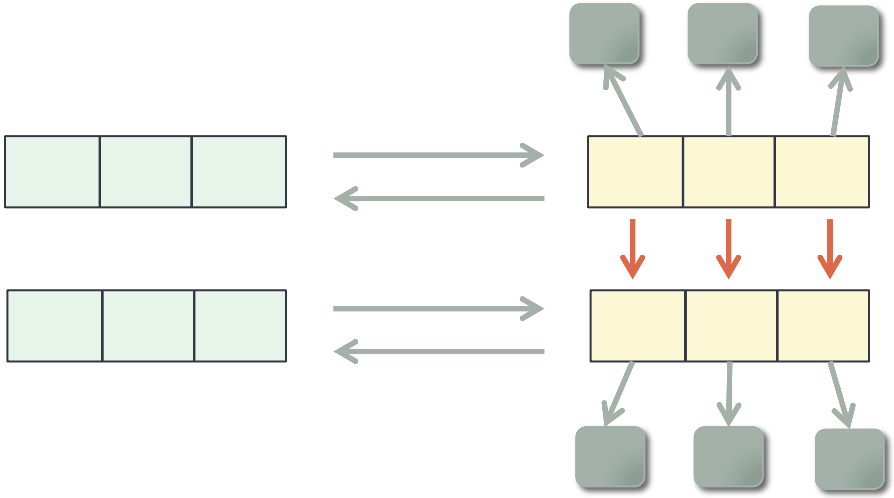
Limitations

```
System.arraycopy(Object src, … Object dest, …)
src, dest: Object[]
```

- Handled in jpf-core



- Handled in jpf-nhandler



Statistics

- `java.lang.String`
  - 24 methods of String are mapped to a method in the String peer
  - Size of `JPF_java_lang_String`: 318 lines
- Created a class that tests every String method that is mapped to a method in the peer
- jpf-nhanlder was used to handle String methods
  - Made jpf-nhandler to delegate all String methods
  - Removed `JPF_java_lang_String`
    - Only delegated “unhandled native” methods

Statistics

class	model class	peer class	effort (code size)
Format	✓	✓	99
NumberFormat	✓	✗	76
DecimalFormat	✓	✓	245
DecimalFormatSymbols	✗	✓	45
DateFormat	✗	✓	75
SimpleDateFormat	✓	✓	177
DateFormatSymbols	✗	✓	36

Total effort: 753 lines of code

Note: model classes and peers are API specific

Future Work

1. Cache JVM objects used in delegation calls (update during gcEnd() notification if ElementInfo was marked as changed)

2. Properly execute clinit() methods with JPF for classes added by JVM→JPF conversion (can be verification relevant code)

3. Implement configuration options (skip, create source stub, create delegating OTF-peer, based on caller or callee packages/classes/ methods)

4. Extend with per-object peers (via ElementInfo attributes) to solve peer state problem

5. Benchmark delegation cases

Thanks!